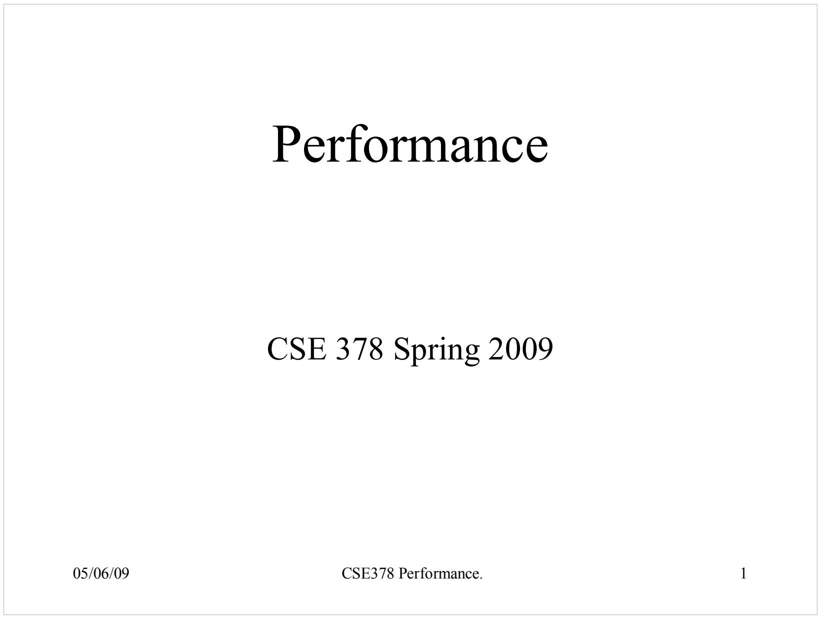
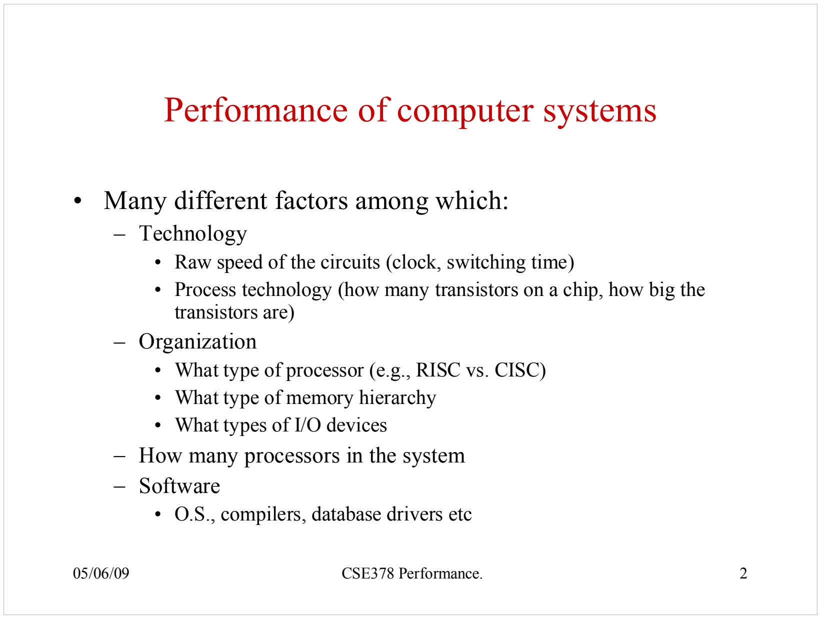


Performance

CSE 378 Spring 2009



Performance of computer systems

- Many different factors among which:
  - Technology
    - Raw speed of the circuits (clock, switching time)
    - Process technology (how many transistors on a chip, how big the transistors are)
  - Organization
    - What type of processor (e.g., RISC vs. CISC)
    - What type of memory hierarchy
    - What types of I/O devices
  - How many processors in the system
  - Software
    - O.S., compilers, database drivers etc

What are some possible metrics?

Traditional measures:

- Raw speed (peak performance = clock rate)
- ***Execution time*** (or response time): time to execute a program from beginning to end.
  - Need benchmarks for integer dominated programs, scientific, graphical interfaces, multimedia tasks, desktop apps, utilities etc.
- ***Throughput*** (total amount of work in a given time)
  - measures utilization of resources (good metric when many users: e.g., large data base queries, Web servers)
  - Improving (decreasing) execution time will improve (increase) throughput.
  - Most of the time, improving throughput will decrease execution time

What are some possible metrics?

Recently:

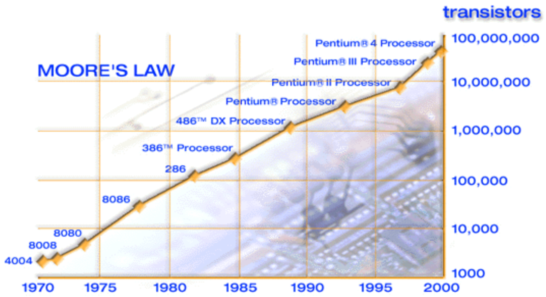
- Measures that concern power
  - Watts = joules / second
  - Energy per instruction = joules / instruction executed

- Why be concerned about power?
  - Battery life in portable devices
  - Heat dissipation issues
  - Server rooms are most constrained by their cooling capacity
  - Dense clusters can be constrained by the ability to route enough power into the installation and/or to the individual processors

CPU Execution Time

Execution_time = (#insts executed) * CPU * (time/cycle)

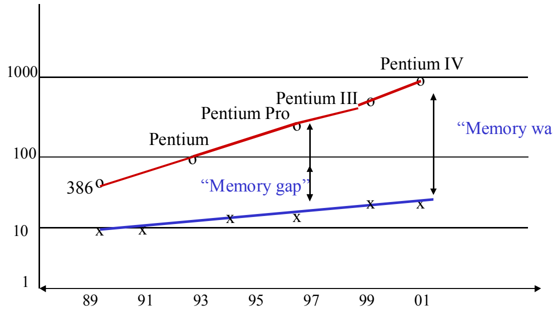
Moore’s Law



Courtesy Intel Corp.

Processor-Memory Performance Gap

- x Memory latency decrease (10x over 8 years but densities have increased 100x over the same period)
- o x86 CPU speed (100x over 10 years)



Comparing Processors Isn’t Straightforward

- Different architectures have different instruction sets
  - Can’t run the same set of (machine) instructions on both
- Even different models in the same architecture may have a complicated relationship
  - Model A’s multiply is 6 times faster than model B’s
  - Model A’s add is 3 times faster than model B’s
  - Model A’s memory system is 8 times faster than model B’s
- But, we really want to compare performance across processors…

Comparing Performance

- The "right measure" is execution time
  - Take some C program, compile, link and run on both processors
  - Measure the time it takes from start to end of the execution

- Notice that this means we are evaluating the compilers as well as the processors
  - Is that reasonable?

- If we're not careful, we might be measuring other things as well
  - E.g., speed of IO devices

Execution time Metric

- Execution time: inverse of performance

  $Performance_A = 1 / (Execution\_time_A)$

- "Processor A is faster than Processor B"

  $Execution\_time_A < Execution\_time_B$

  $Performance_A > Performance_B$

- Relative performance (a computer is "n times faster" than another one)

  $Performance_A / Performance_B = Execution\_time_B / Execution\_time_A$

Definition of CPU execution time

CPU execution_time = (#cycles) * (time per cycle)

- (#cycles) depends on program, compiler, and input
- (time per cycle) is the inverse of clock rate
  - Depends on the processor's implementation
  - Clock rate measured in MHz or GHz

Another form of the equation

CPU execution_time =
(#insts executed) * (cycles / instruction) * (time/cycle)

- (cycles / instruction) is called CPI
- CPI depends on processor's implementation:
  - CPI = 1 "Single cycle"
  - CPI > 1 Some instructions require more than one cycle
  - CPI < 1 Some form of parallel execution

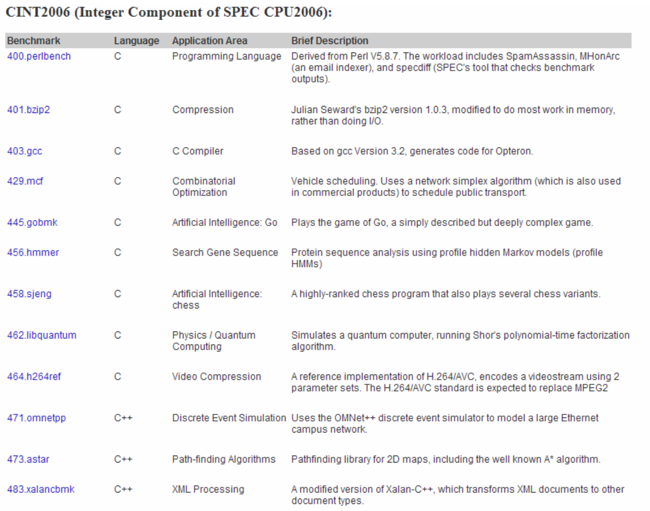
How to Improve Performance?

CPU execution_time = (#insts executed) * CPU * (time/cycle)

- Reduce (#insts executed) : better compilers
- Reduce (time/cycle) : higher clock rates or better processor implementations
- Reduce CPI : more internal parallelism in processor implementation
  - *Pipelining, Superscalar, multi-threaded, multi-core*

Benchmarks

- Benchmark: workload representative of what a ***system*** will be used for
- Industry benchmarks
  - **SPECint** and **SPECfp** industry benchmarks updated every few years,
  - Linpack (Lapack), NASA kernel: scientific benchmarks
  - TPC-A, TPC-B, TPC-C and TPC-D used for databases and data mining
  - Other specialized benchmarks (Olden for list processing, Specweb, SPEC JVM98 etc…)
  - Benchmarks for desktop applications, web applications are not as standard
  - Beware! Compilers (command lines) are super optimized for the benchmarks



CINT2006 (Integer Component of SPEC CPU2006):

Benchmark	Language	Application Area	Brief Description
400.perlbench	C	Programming Language	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	C	Compression	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
445.gobmk	C	Artificial Intelligence: Go	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	C	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	C	Artificial Intelligence: chess	A highly-ranked chess program that also plays several chess variants.
462.libquantum	C	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	C	Video Compression	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
471.omnetpp	C++	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	C++	Path-finding Algorithms	Pathfinding library for 2D maps, including the well known A* algorithm.
483.xalancbmk	C++	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.

How to summarize benchmark performance

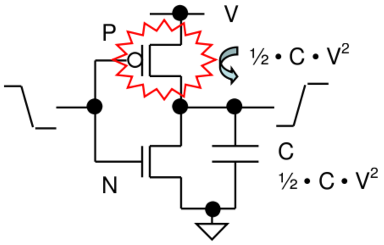
- n programs in the benchmark suite. What is the relative performance "overall"?

- A number of alternatives:
  - **arithmetic mean** of execution **times:**
    - $(\Sigma exec\_time_i) / n$
  - **harmonic mean** of **rates:**
    - $n/ (\Sigma\ 1/rate_i)$
  - **geometric mean** of **rates:**
    - $(\Pi\ rate_i)^{1/n}$

Power

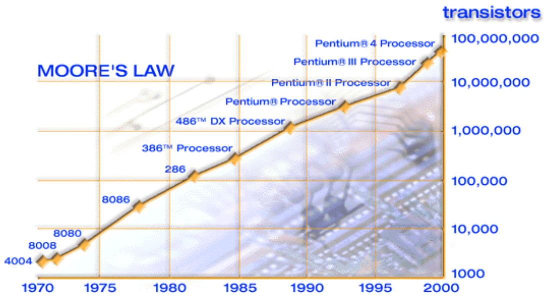
EPI = Joules / instruction = Watts / IPS

Dynamic Power Dissipation

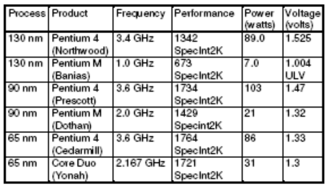

V
P
$\frac{1}{2} \cdot C \cdot V^2$
C
$\frac{1}{2} \cdot C \cdot V^2$
N

CMOS Inverter

Moore’s Law

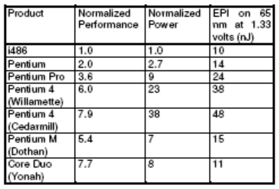


Courtesy Intel Corp.



Process	Product	Frequency	Performance	Power (watts)	Voltage (volts)
130 nm	Pentium 4 (Northwood)	3.4 GHz	1342 SpecInt2K	89.0	1.525
130 nm	Pentium M (Banias)	1.0 GHz	673 SpecInt2K	7.0	1.004 ULV
90 nm	Pentium 4 (Prescott)	3.6 GHz	1734 SpecInt2K	103	1.47
90 nm	Pentium M (Dothan)	2.0 GHz	1429 Specint2K	21	1.32
65 nm	Pentium 4 (Cedarmill)	3.6 GHz	1764 SpecInt2K	86	1.33
65 nm	Core Duo (Yonah)	2.167 GHz	1721 SpecInt2K	31	1.3

Table 2: Performance and Power of Intel Microprocessors, 130 nm to 65 nm



Product	Normalized Performance	Normalized Power	EPI on 65 nm at 1.33 volts (nJ)
i486	1.0	1.0	10
Pentium	2.0	2.7	14
Pentium Pro	3.6	9	24
Pentium 4 (Willamette)	6.0	23	38
Pentium 4 (Cedarmill)	7.9	38	48
Pentium M (Dothan)	5.4	7	15
Core Duo (Yonah)	7.7	8	11

Table 3: EPI of Intel Microprocessors

http://www.intel.com/pressroom/kits/core2duo/pdf/epi-trends-final2.pdf

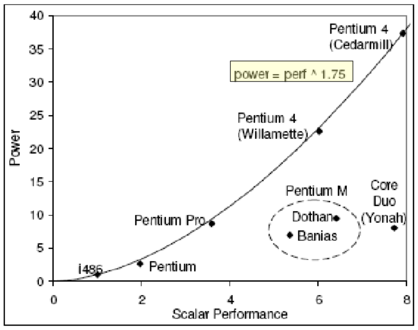


Figure 2: Normalized Power versus Normalized Scalar Performance for Multiple Generations of Intel Microprocessors

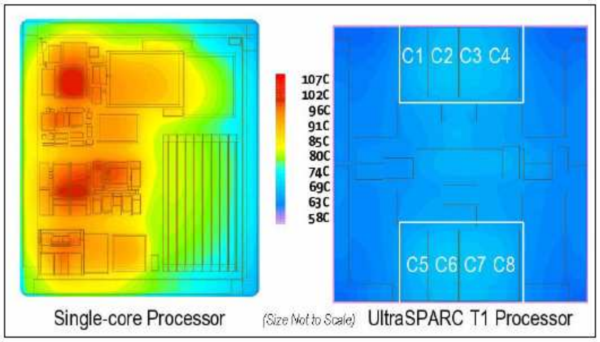


FIGURE 2 **Power Density: Single-Core Processor vs. UltraSPARC T1 Processor**

http://www.sun.com/processors/whitepapers/UST1_pwrsav_v1.0.pdf

Parallelism

S(∞) = 1/f (Amdahl's Law)

Amdahl's Law (Parallel Processor Speedup)

- S(P) is parallel speedup using P processors
  - (sequential execution time) / (parallel exec time using P processors)
- Assume:
  - fraction f of the application's execution is “inherently sequential”
  - fraction (1-f) can be perfectly parallelized
- S(P) = 1 / (f + (1-f)/P)
- S(∞) = 1/f
  - For example, if 20% of your program is inherently sequential, the maximum possible parallel speedup is at 5

- *What fraction of rendering a web page is inherently sequential?*
- *What fraction of Google's workload is inherently sequential?*

What Next?

- We'll look at parallelism
- First, pipelining – a simple approach to speeding up the data path
- Then, “instruction level parallelism”: more aggressive techniques for to allow execution of more than one instruction at a time
- Both of the above preserve the sequential semantics of the ISA
- Multi-core changes the ISA (and makes exploiting parallelism the software's problem...)